

Overcoming the Memory Bottleneck in Suffix Tree Construction

Martin Farach*

Bell Labs & Rutgers University

Paolo Ferragina†

Max Planck Institut für Informatik

S. Muthukrishnan‡

Bell Labs

Abstract

The suffix tree of a string is the fundamental data structure of string processing. Recent focus on massive data sets has sparked interest in overcoming the memory bottlenecks of known algorithms for building suffix trees.

Our main contribution is a new algorithm for suffix tree construction in which we choreograph almost all disk accesses to be via the `sort` and `scan` primitives. This algorithm achieves optimal results in a variety of sequential and parallel computational models. Two of our results are:

- In the traditional external memory model, in which only the number of disk accesses is counted, we achieve an optimal algorithm, both for single and multiple disk cases. This is the first optimal algorithm known for either model.
- Traditional disk page access counting does not differentiate between random page accesses and block transfers involving several consecutive pages. This difference is routinely exploited by expert programmers to get fast algorithms on real machines. We adopt a simple accounting scheme and show that our algorithm achieves the same optimal tradeoff for block versus random page accesses as the one we establish for sorting.

1 Introduction

Many rapidly growing stores of data, both scientific and commercial, are being accumulated in data warehouses and digital libraries. These data are typically far too large to reside in main memory; they reside on “external memory” such as disks, or in some cases on tapes. There is a general consensus that these data

are an asset, and we must therefore find ways to efficiently archive, analyze and extract value from these data. Fundamental new algorithmic issues arise when we look at data which resides on disks, because the memory is a potential bottleneck, that is, the performance of algorithms can be dramatically affected by their pattern of disk accesses. For some problems, the issue of carefully structuring memory access patterns to overcome the memory bottleneck has been addressed and is well-understood [16], but for string processing, many issues have not been settled [4].

The string processing tasks which are among the most prevalent in systems for massive data set manipulation include compression, indexing and information retrieval. Each of these tasks relies on string processing primitives, many of which are most efficiently solved by constructing the suffix tree data structure on the underlying strings. While the suffix tree data structure plays the pivotal rôle in many other applications [10], we are concerned with the case where the string—and the suffix tree—do not fit into the main memory. In this paper, we give algorithms for suffix tree construction that are optimal in different ways of counting the memory accesses and modeling the memory bottleneck (in sequential and parallel models). In what follows, we describe memory access accounting schemes in detail before presenting our technical results.

1.1 Memory Issues and Models

In standard algorithmic design, the assumption is that all memory accesses cost unit time. This does not account for differences in the memory hierarchy. In order to capture such differences, consider a *Disk Access Machine (DAM)*, in particular, the one introduced by Vitter and Shriver [17]. A DAM¹ consists of a processing unit, an internal memory of size M and a large external memory which is partitioned into transfer units, called *disk pages*, each containing B items; note that $M < N$ and $1 \leq B \leq M/2$. We must

*Bell Labs, Room 2C-414, 600 Mountain Ave., Murray Hill, NJ 07974. (farach@research.bell-labs.com)

†Max Planck Institut für Informatik, Im Stadtwald, 66123 Saarbrücken (Germany). Supported by EU ESPRIT LTR Project N. 20244 (ALCOM-IT), research grant of University of Pisa (1998), and research grant of Italian Ministry of Research “Tecniche formali per sistemi software.” (paolo@mpi-sb.mpg.de)

‡Bell Labs, Room 2C-357, 600 Mountain Ave., Murray Hill, NJ 07974. (muthu@research.bell-labs.com)

¹This model has also been called the *Two-level memory model* [17], and some variants include the *hierarchical memory model* [1, 3], the *block transfer model* [1], etc.

now decide how to analyze algorithms, that is, what operations to count.

Counting Disk Accesses. Disk accesses are believed to be the dominating factor in the running time of many algorithms because of the relative speed of the disk versus the main memory access time [13]. Accurate disk models are complex [14, 17, 16, 15], and it is virtually impossible to exploit all the fine points of disk characteristics systematically, either in practice or for algorithmic design. A simplification on a DAM is to count the total number of disk accesses (I/Os) performed by the various operations. This accounting scheme does not accurately predict the running time of algorithms on real machines. However, it is a workable approximation for algorithm design, and it has gained tremendous popularity [16].

Counting Random I/Os. Since accessing a page from the disk in most cases decreases the cost of accessing the page succeeding it, “block” I/Os are less expensive per page than “random,” that is, non-adjacent, I/Os. This dictum is well believed, and it is routinely exploited by expert programmers in practice to get better performance. There is no well established way to account for this difference between the random and block I/Os except to introduce new parameters into the accounting and the analysis (such as the average latency to access the first page and the incremental bandwidth etc.) which complicates the understanding of inherent tradeoffs, and makes the design of algorithms cumbersome.

We adopt the following simple accounting scheme here to differentiate random and block I/Os. Let c be a constant and define a *Block* to be a sequence of cM adjacent memory locations in main memory or on disk, that is, cM/B pages. A *Block Transfer* is any I/O which involves moving all the pages of a block between main memory and disk. Any page I/O which is not part of a block transfer is said to be a *random* I/O transfer. We augment the analysis of our DAM algorithms so that for any algorithm, we separately count the total (block and random) number of page I/Os as well as the number of random page I/Os (we do not explicitly count the block I/Os) and present both page counts as the complexity of the algorithm. See Section 2 for a justification of this scheme.

1.2 Our Results

Given a string $S \in \Sigma^N$, the *suffix tree* T_S of S is the compacted trie of all the suffixes of $S\mathcal{Y}$, $\mathcal{Y} \notin \Sigma$. In the comparison model, where $|\Sigma|$ is unbounded, the suffix tree can be built in $O(N \log N)$ time [18, 12]. Since the

suffix tree represents all suffixes of a string in sorted order, this is optimal for the comparison model.² On the DAM, the number of total I/Os needed for sorting is $\Omega(N/B \log_{M/B} N/M)$ [2], and this also holds for suffix tree construction. For new bounds on the random vs total I/O tradeoff for sorting, see Section 2.

Our main contribution is an approach to suffix tree construction in which we choreograph all external memory accesses using, essentially, only sorts and scans as primitives. By using optimal sorting routines and adapting the scan operations appropriately, we can apply our approach uniformly to several sequential, parallel and memory models, and thus obtain the first known optimal algorithms therein. We get the following specific results.

First we consider the traditional scheme of counting total page I/Os. We obtain an optimal algorithm for suffix tree construction that uses $O(N/B (\log_{M/B} N/M))$ I/Os. The previously known algorithms, which involved PRAM simulations, were suboptimal by a polylogarithmic factor. Further, our algorithm was designed so that even in the random/total accounting scheme, we achieve asymptotically the same tradeoff between random and total I/Os as sorting (see Section 2), which is optimal. This main algorithm can be extended in many ways. The *Parallel DAM (PDAM)* is a generalization of the DAM model to multiple parallel disks in which the external memory comprises D independent disks and each I/O involves transfer of a page from each disk simultaneously [17]. We obtain the following optimal result:

Theorem 1.1 *Given a string S of length N , the suffix tree of S can be built on the PDAM in $O(N/(DB) (\log_{M/B} N/M))$ I/Os and linear space.*

Previous string processing results on the PDAM have all been suboptimal [4]. Details of our PDAM result are deferred to the full paper.

Trivially, we get the first known I/O-optimal suffix array [11] and SB-tree [9] construction algorithms. Furthermore, many string problems have suffix tree construction as their I/O bottleneck [10]. All these now have efficient implementations on the (P)DAM. Also, we obtain improved results on parallel machines (details in the full paper):

Theorem 1.2 *Given a string S of length N , the suffix tree of S can be built on the EREW PRAM model in $O(\log^4 N)$ time, $O(N \log N)$ work and linear space.*

²An $O(N)$ algorithm exists for binary alphabets [18, 12] and for integer alphabets [7].

Theorem 1.3 *Given a string S of length N distributed evenly among p processors of a BSP machine, the suffix tree of S can be built in $O(\frac{N \log N}{p})$ internal computation time and $O(\frac{\log^3 N \log p}{\log h})$ communication rounds using h -relations where $h = \Theta(N/p)$.*

1.3 Technical Overview

There are two known approaches to suffix tree construction: the classical one-suffix-at-a-time approach, and the recent divide-and-conquer approach. In trying to give an I/O efficient algorithm with the classical approach, one could imagine extending the suffix tree in batches. However, such an approach would require looking up strings in data structures on partially built suffix trees. While it is possible that this could be done in an I/O efficient manner, these I/Os would certainly be random, rather than block, leading to a slower algorithm.

We take the divide-and-conquer approach. The technical challenge is to merge so-called *odd* and *even* trees, T_o and T_e , respectively, using only sorts and scans. We do so by showing new structure in these trees. While the structure of suffix trees has been important for designing efficient algorithms in the past, we show that the partial suffix trees T_o and T_e themselves have substantial structure, such as zipper teeth, which can be exploited algorithmically.

Outline The paper is organized as follows. In Section 2, we discuss the issues involved in block versus random I/O. In Section 3 we present notation and tools. In Section 4 we present the main part of the algorithm for the DAM.

2 Block versus Random I/O

Disk access delays depend on seek time, rotational latency and transfer time. Accessing disk pages in blocks amortizes the latency involved in moving the head on a disk. In a typical seek on a modern disk, the arm moving time approximately equals the rotational time. Thus, for example, writing a single page, then, after some computation, writing the next page on a disk would incur a hefty penalty associated with rotational latency. Furthermore, if the computation takes place in a multi-user environment, the arm may move between disk I/Os, further complicating matters. Thus, we consider simple models of block disk I/O in which all the contiguous disk pages must be read or written as an atomic event.

Let C be the size of a block to be read or written, and let B be, as before, the size of a page in the DAM. A page is *random* if it is not accessed within

a block of C/B pages, and *block* otherwise. Let $b(N)$ be the number of block I/Os performed by an algorithm on an input of size N , and $r(N)$ the number of random I/Os. Then the total number of I/Os is $t(N) = r(N) + b(N)C/B$, and we can characterize the I/O profile of an algorithm by the pair $(t(N), r(N))$. Different choices of C naturally give different profiles, and of course, specifying B and C is different from simply enlarging the page size.

So the question is, how do we choose C ? We would like to make C significantly bigger than B , to amortize latency over block accesses. Ideally, the average access time for an item on disk should be around the same as item in memory. First, notice that disk bandwidth is large enough to make this feasible, and that bandwidth is improving at least as quickly as memory access times. Also, seek time on disk is decreasing much slower than other system parameters. For example, disk seek time is decreasing by about 10% per year, while memory access time is decreasing by about 70% per year [13]. Therefore, the number of items which must be accessed on disk in order to amortize the cost is increasing exponentially, with a doubling time of roughly 18 months [6]. Similarly, the size of main memory is also doubling approximately every 18 months. Consequently, it is natural to set our block size to be $C = cM$, for a constant c . On real systems, these doubling times will never match, but the two growth rates are similar enough so that qualitatively, we can say that our model of block accesses captures the behavior of disks over a substantial window of time.

This choice has a further advantage. In any model, introducing a new parameter – latency, block size, etc. – is undesirable for understanding issues in algorithmic design. Taking $C = cM$ means that we can simulate any algorithm with any choice of c with any other choice of c' with a constant increase in the total I/Os and with no increase in the number of random I/Os, our accounting method is parameter-free. Although this accounting scheme does not capture all the aspects of block vs random I/Os, it lower bounds the benefit accrued from block access in general, by only considering the extremal case when the main memory is nearly “flooded” with each block I/O access.

Sorting. Given this new accounting scheme, we reconsider the I/O profile of sorting. Simple 2-way merging induces no random I/Os, but gives $O(\frac{N}{B} \log_2 \frac{N}{M})$ total I/Os. If we performed M/B -way merging, we would get $O(\frac{N}{B} \log_{M/B} \frac{N}{M})$ total I/Os, all of which would be random. The natural question arises as

to whether we can achieve the same number of total I/Os as the multi-way mergesort without such a heavy penalty in random I/Os. The following shows that this is not possible, that there is no smooth trade-off between total and random I/Os, and thus that our only real choices are 2-way or M/B -way merging.

Lemma 2.1 *Any DAM algorithm for sorting N numbers using $o(\frac{N}{B} \log_{M/B} \frac{N}{M})$ random I/Os needs $\Omega(\frac{N}{B} \log_2 \frac{N}{M})$ total I/Os.*

Proof (Sketch): Consider a decision tree where every node is either a *random-node* having fan-out $\binom{M}{B}$, corresponding to a random I/O, or a *block-node* having fan-out $\binom{M}{C}$, corresponding to a block I/O. Consider now the class of sorting algorithms executing r random and b block I/Os. They can be represented as decision trees where any path has r random- and b block-nodes, and they consists of at least $N!/(M!)^{N/M}$ leaves, where this is the number of classes of permutations such that any permutation in an equivalence class can be generated from any other by a scan. Following an argument similar to that in [2], we obtain $rB \log_2(M/B) + Cb \log_2(M/C) \geq N \log_2(N/M)$. Since $C = cM$, it follows that $b \geq (N/M) \log_2(N/M) - r(B/M) \log_2(M/B)$. Now, if r is $o((N/B) \log_{M/B}(N/M))$, then the relation above becomes $b = \Omega((N/M) \log_2(N/M))$, which is exactly the bound of 2-way mergesort. ■

3 Preliminaries

Structure of Suffix Trees. Let $S[N+1] = \text{¥}$ be a special end-of-string marker. We will represent suffix trees as follows. Leaf l_i represents suffix $S[i, N]$. Each internal node v has associated the length $L(v)$ which is the sum of the edge lengths on the path from the root to v . The *string at v* , denoted $\sigma(v)$, is $S[i, i+L(v)-1]$ where l_i is any leaf below v . The children of node v are stored in a list sorted by the first characters on the edges from v . The parent of v is denoted by $p(v)$. The following well-known lemma gives suffix trees a nice structure.

Lemma 3.1 ([18]) *Let a be a character and α be an arbitrary substring of S . If there is a node v in T_S such that $\sigma(v) = \alpha a$, then there is a node w in T_S such that $\sigma(w) = \alpha$.*

Consequently, we can define, for every node v in a suffix tree, the *suffix link* $sl(v) = w$, where v and w are defined as in Lemma 3.1. Notice that $sl(\cdot)$ links form a tree rooted at the root of T_S . The depth of any node v in this $sl(\cdot)$ tree is then just $L(v)$.

Let $\text{lcp}(\alpha, \beta)$ be the *length of the longest common prefix* of two strings α and β . Let $\text{lca}(v, w)$ be the *least common ancestor* of any two nodes v and w in a tree. The property of suffix trees often exploited algorithmically is the following relationship between lcp in S and lca in T_S : $\forall v, w \in T_S, \quad \text{lcp}(\sigma(v), \sigma(w)) = |\sigma(\text{lca}(v, w))|$.

3.1 General Tools for Computations on the DAM

In this subsection we review some known results [5] and propose simple, new tools that are used subsequently as basic blocks in our algorithm. We will be reducing the I/O complexity of various problems to the I/O complexity of sorting, with its tradeoff of random versus total I/Os, so for shorthand, throughout the rest of the paper, we will say that any algorithm with the same I/O complexity and tradeoff as sorting N items has *sorting I/O complexity*.

1. Given a tree T of size N , we can answer a batch of $K \leq N$ lca-queries in sorting I/O complexity [5]. Equivalently, let A be an array of numbers and define $\text{rmq}_A(i, j) = \min_{i \leq k \leq j} \{A[k]\}$ as the *range minimum query* from i to j in A . We can answer K range minima queries in sorting I/O complexity.
2. Given a tree T of size N , we can construct its Euler Tour $\text{ET}(T)$ and determine the depth of each of its nodes in sorting I/O complexity [5].
3. Given a string $S[1, N]$ stored contiguously on disk, we can retrieve K arbitrary characters from it in sorting I/O complexity.
4. Given a tree $T = (V, E)$ of size N and a set of marked nodes $M \subseteq V$, let $D : V \rightarrow M \cup \{0\}$ be any function such that $D(v) = w$ if w is a marked descendant of v , and $D(v) = 0$ if v has no marked descendant. Then, we can compute $D(\cdot)$ in sorting I/O complexity by first computing $\text{ET}(T)$ and then properly scanning it. In particular, for each node, we can find one of its descendant leaves in sorting I/O complexity.

Theorem 3.2 *If N is the size of the tree, array or list to be manipulated and K is $O(N)$, then all the problems listed above can be solved in sorting I/O complexity.*

3.2 Subroutines

We conclude with a few subroutines which use some of the tools above.

Skeleton Trees: Let T be an N node compacted trie built on a string set Δ and let $\Delta' \subset \Delta$. Call $T_{\Delta'}$ the *skeleton tree* of T with respect to Δ' .

Theorem 3.3 *Given a set Δ of N strings and its compacted trie T , the skeleton tree $T_{\Delta'}$ can be constructed on any subset of strings $\Delta' \subset \Delta$ in sorting I/O complexity.*

Merging Uncompacted Tries: Merging two uncompacted tries in linear time is trivial in internal memory by a coupled visit which advances in both tries as long as the corresponding characters match, and otherwise outputs the “lexicographically smaller” subtree. On the DAM, such a visit is implemented by computing first the Euler Tour of the two tries and then simply scanning them. The bottleneck is to compute the Euler Tours, which has sorting I/O complexity (Theorem 3.2).

Theorem 3.4 *Two uncompacted tries with a total of N nodes can be merged in sorting I/O complexity.*

Computing Suffix Links: In a suffix tree, we are guaranteed that a suffix link pointer is defined for every node [12]. These links can be efficiently computed as follows.

Theorem 3.5 *Given a suffix tree with a total of N nodes, its suffix links can be computed in sorting I/O complexity. $L(v)$ can be computed for each node v in the same I/O complexity.*

Proof: For every internal node v , pick two descendant leaves l_i and l_j such that $\text{lca}(l_i, l_j) = v$. Let $w = \text{lca}(l_{i+1}, l_{j+1})$. Since $\text{lcp}(S[i, N], S[j, N]) = 1 + \text{lcp}(S[i+1, N], S[j+1, N])$, then $sl(v) = w$. Now, the computation of each suffix link is independent of the others, so we can form a batch of $O(N)$ lca -queries. Recall also that the depth of any node v in the $sl(\cdot)$ tree is just $L(v)$, so we finish by computing the depth of every node. Each of these steps can be implemented to take sorting I/O complexity (Theorem 3.2). ■

3.3 Algorithm Outline

Our algorithm uses the *odd/even divide-and-conquer* approach [8, 7], which consists of three steps: building the *odd tree*, building the *even tree*, and merging the two trees. Since the first two steps are relatively straightforward, even on the DAM, we will skip the details of building these trees in this abstract and devote the rest of the paper to merging the two trees.

It is this last step which departs substantially from the merging algorithms in [8, 7]. We will need to show how to exploit the structure of odd and even trees to batch character queries in order to establish our result.

Building the odd tree: Consider the set of all the suffixes of S starting in odd positions, $Odd = \{S[2i+1, N+1] \mid 0 \leq i \leq N/2\}$, and let the *odd tree* T_o be the compacted trie built on Odd . We build this tree recursively, and conclude:

Lemma 3.6 *If $C(N)$ is the I/O-complexity taken by our algorithm to build the suffix tree of a string $S[1, N]$, then T_o can be built in $C(N/2)$ plus the I/O complexity of sorting N numbers.*

Building the even tree: The *even tree* T_e is simply the compacted trie built on all suffixes of S beginning at the even positions. We can build this tree directly from the odd tree and show that:

Lemma 3.7 *Given a string $S[1, N]$ and its odd tree T_o , the even tree T_e can be constructed in sorting I/O complexity.*

Merging: We must now merge T_o and T_e , in order to get T_S . We show how to do this in sorting I/O complexity in the following section, thus concluding the algorithm.

4 Merging the Odd and Even trees

While two distinct algorithms exist for merging odd and even trees [8, 7], both require that arbitrary characters of the underlying string be available throughout the merging. These character lookups have no locality of reference, and thus elicit many random disk accesses. In this section, we propose a new way to merge odd and even trees based only on sorting and scanning, thus achieving our optimal bounds for total and random I/Os. We will first restate the problem of Odd/Even tree merging, thereby eliminating some extraneous considerations, before tackling the core of the problem.

4.1 Merge Nodes

For simplicity, we will identify each node v with its string $\sigma(v)$ and say that node v in one tree equals node w in the other tree if $\sigma(v) = \sigma(w)$. Define a *merge node* of T_o , resp. T_e , to be any node v whose occurrence in T_S has both odd and even descendants. Notice that if v is a merge node, then all of its ancestors are also merge nodes. Identifying the merge nodes turns out to be equivalent to merging the two trees T_o and T_e .

Lemma 4.1 *Given a string $S[1, N]$, its odd tree T_o and its even tree T_e such that every merge node in each tree is flagged, T_S can be constructed in sorting I/O complexity.*

Proof: First, construct $ET(T_o)$ and $ET(T_e)$ and retrieve all the first characters labeling the tree edges in sorting I/O complexity (Theorem 3.2). Then, merge $ET(T_o)$ and $ET(T_e)$ by a coupled visit guided by the flagged merge nodes. Specifically, consider the leftmost child v_o of the root of T_o and the leftmost child v_e of the root of T_e . If both these nodes are merge nodes, then recursively merge the sub-Euler tour of $ET(T_e)$ from the first to last occurrence of v_e with the sub-Euler tour of $ET(T_o)$ from the first to last occurrence of v_o and output the result. If neither is a merge node, there are two further sub-cases. If the edges leading to v_o and v_e , the nodes we are currently processing, have distinct first labeling character, then output the sub-Euler tour of whichever subtree comes lexicographically first. Otherwise they share the first labeling character and we “partially” merge these edges by creating a new node w , with unspecified $L()$ value, with two children v_o and v_e . This changes the Euler tour by a constant number of new edges, in particular, the ones connecting w with v_e and v_o , but the sub-Euler tours descending from v_o and v_e remain unchanged.

Finally, consider the case in which only one node between v_o and v_e is a merge (flagged) node. Here too, we distinguish two sub-cases. If the edges leading to v_o and v_e have distinct first labeling character, then output the sub-Euler tour of whichever subtree comes lexicographically first. Otherwise enter in a loop, because we are guaranteed that on the edge leading to v_o some nodes of T_e must be inserted. The first of these, of course, is v_e and the others form a path. Keeping this in mind, we install on this edge all the flagged nodes in T_e descending from v_e by a simple scan of the sub-Euler Tour descending from v_e . When we reach the last flagged node, we exit from the loop and resume merging with the case above where the current v_e and v_o are not merge nodes.

The whole procedure has sorting I/O complexity. The only checked characters are the first ones labeling the edges of T_o and T_e , which were initially retrieved in sorting I/O complexity. The depth of the newly installed nodes (e.g. node w above) is finally determined by applying Theorem 3.5. ■

Hence, we are left with the problem of finding merge nodes in T_o and T_e . We consider further simplifications below.

4.2 Anchor Nodes and Side Trees

Anchor nodes and *side trees* were introduced in [8]. An *anchor pair* is a pair of nodes v_o, v_e in T_o and T_e , respectively, such that $\sigma(v_o) = \sigma(v_e)$; each such v_o and v_e is an *anchor*. All anchor nodes and their ancestors are merge nodes. Finding anchor nodes was part of the randomized PRAM algorithm in [8], but that algorithm does not meet our purposes. Below, we show how to exploit the structure of odd and even trees to find anchor nodes in the DAM model I/O-efficiently.

In particular, to find anchor nodes, we will use suffix links. Recall that suffix links $sl(\cdot)$ define a tree which can be computed in sorting I/O complexity (Lemma 3.5). Unfortunately, neither T_o nor T_e is a suffix tree, and so they are not guaranteed to have well-defined suffix links. If we apply the procedure from Lemma 3.1 to T_o and T_e jointly, the “suffix links” of nodes in the odd tree point to nodes in the even tree, and vice versa. These links form two trees, one rooted at the root of the odd tree and the other rooted at the root of the even tree. We call these trees L_o and L_e , respectively.

Each edge in L_o and L_e can be labeled with a single character as follows. If $sl(v) = w$ and $\sigma(v) = a\sigma(w)$, for $a \in \Sigma$, label the edge (v, w) with character a . Then all the children of a node in either L_o or L_e are labeled with different characters. The following lemma relates L_o and L_e with anchor pairs and is the basis of our algorithm.

Lemma 4.2 *If v_o, v_e is an anchor pair in T_o and T_e , then one occurs in each of L_o and L_e . Furthermore, the path from the appropriate root to each of v_o and v_e is labeled with the reversal of $\sigma(v_o)$ ($= \sigma(v_e)$).*

Proof: Every time an $sl(\cdot)$ link is traversed, the parity of the length of the string at that node changes. Since $\sigma(v_o) = \sigma(v_e)$, they cannot be in the same tree. Additionally, a simple inductive proof shows that the characters labeling the path from the root, of either L_o or L_e , to v_o are the reversal of $\sigma(v_o)$, and similarly for v_e . ■

We conclude that if we treat L_o and L_e as (uncompacted) tries, and we merge them, then the nodes which are merged are anchor node pairs. Consequently, we merge these two uncompacted tries via Theorem 3.4 and conclude:

Lemma 4.3 *Given a string $S[1, N]$, its odd tree T_o and its even tree T_e , the set of anchor-node pairs can be determined in sorting I/O complexity.*

While all the ancestors of anchor nodes are merge nodes, some of the descendants may also be merge nodes. Below, we characterize this set of merge nodes and explain how to efficiently find them on the DAM, thus completing our algorithm.

Let a *side tree* be a maximal component of nodes such that no node in a side tree has an anchor descendant. Let a_o, a_e be an anchor pair, and let s_o, s_e be two side trees with root r_o and r_e , respectively, such that r_o 's (resp. r_e 's) parent is a_o (resp. a_e), and such that the two edges (a_e, r_e) and (a_o, r_o) share (at least) the first labeling character. We call such a pair of subtrees s_o and s_e a *side tree pair*. Once we have computed all anchor nodes, we can easily find all side tree pairs in T_o and T_e by retrieving all the pairs of side trees such that: (i) they hang off an anchor pair, and (ii) the edges (a_o, r_o) and (a_e, r_e) share the first labeling character. Thus, we have:

Lemma 4.4 *Once all anchor pairs have been flagged, all side tree pairs can be found in sorting I/O complexity.*

Proof: Build $ET(T_o)$ and $ET(T_e)$ (Theorem 3.2) and scan them, provided that anchor pairs have been flagged and the first character labeling each edge in T_o and T_e have been retrieved (Theorem 3.2). ■

While side tree pairs do not contain anchor nodes, they still may contain merge nodes. However, merge nodes in side tree pairs have the nice structure that they form a path [8]. Since all ancestors of a merge node are merge nodes, we can think of merging side trees as closing a *zipper*. The nodes of the merge path interdigitate like the teeth of a zipper, except that the nodes need not alternate in lockstep like the teeth of a zipper. Finding the merge nodes in s_o and s_e now simply consist of finding the teeth of the side trees since the other merge nodes are their ancestors.

4.3 Finding the Zipper Teeth

Rather than finding the teeth directly, we will first find a leaf below the teeth of each side tree. We call such leaves *pull nodes*. In effect, we will use a pair of pull nodes to close the zipper between side tree pairs. In this subsection, we will show that it suffices to find pull nodes, because we can easily derive the teeth from them. This algorithm will be a variant on the *Unmerging* step of [7], however, the added structure and properties of pull nodes will allow us to perform this step efficiently on a DAM.

Suppose then that we have computed all pull nodes, that is, for each side tree pair, we have a pair of leaves, one below the odd teeth and one below the

even teeth. By Theorem 3.2, we can mark all nodes which have pull descendants as “merge” nodes, and use Lemma 4.1 to merge T_o and T_e into some tree T' . Notice that we are misapplying Lemma 4.1, since this lemma is only guaranteed to work with correctly marked merge nodes. However, if we apply this procedure to our trees, we will merge correctly as far down as the anchor nodes and below that, we will only be doing zipper merges. So while T' is not T_S , at least it is a tree.

After having misapplied Lemma 4.1, we will misapply Lemma 3.1 to compute “suffix links” with one small difference. For any node v , rather than picking an arbitrary pair of leaves descending from v , as called for by Lemma 3.1, we pick one odd leaf and one even leaf. Let $L^*(v)$ be the depth of v in the “suffix link” tree computed by this method, and let $L(v)$ be the length of the node in its original tree, that is, $L(v)$ is $|\sigma(v)|$ in the odd tree, if v is an odd node, and in the even tree otherwise. Notice that any ancestor of a pull node will always have odd and even leaf descendants, and we can find such a pair in sorting I/O complexity (Theorem 3.2). No other node could be a merge node, and need not be considered in our computation. Even though we misapplied the two Lemmas above we can still identify the merge nodes in the side trees by a simple scan, since for any node u in T' , u is a merge node iff $L(u) = L^*(u)$ (proof omitted).

Lemma 4.5 *Given the pull nodes for every side tree pair, we can compute merge nodes in sorting I/O complexity.*

4.4 Finding the Zipper Pulls

We are left with the problem of detecting the pull nodes for each side tree pair. We will present this algorithm in three steps: first, we will give a linear time internal memory algorithm, and then an I/O-inefficient algorithm for computing pull nodes. Finally, we will give an I/O-efficient algorithm for finding pull nodes which takes sorting I/O complexity, thus completing the presentation.

An internal memory algorithm: In internal memory, the task of finding pull nodes is easy. Suppose we have a side tree pair s_o, s_e hanging off of an anchor pair a_o and a_e . Let $e_o = (a_o, r_o)$ be the edge incident on s_o and $e_e = (a_e, r_e)$ be the similarly defined edge between a_e and s_e . Let $|e_o| = |\sigma(r_o)| - |\sigma(a_o)|$ and similarly define $|e_e|$. Suppose $|e_o| > |e_e|$, and treat the other case similarly. Introduce a new unary node u in e_o and set $L(u) = L(r_e)$. Now, if r_e is a merge node, the pair u, r_e forms an anchor pair. Node u has

one child, but r_e has more than one. We can determine if there is a side tree pair below u, r_e as done before: check if the first character on the edge (u, r_o) matches the first character on one of the edges below r_e . If so, we have a new side tree pair, and can continue as before. Otherwise, we pick any leaf below u and any leaf below r_e as the pull pair.

If r_e is not a merge node, then the side tree pair s_o, s_e contains no merge nodes, and any pair of leaves, one from s_o and the other from s_e , will work as pull nodes. Therefore, it can't hurt to continue the algorithm as if r_e were a merge node, since in this case, anything we do is right.

This gives a linear time algorithm for finding pull nodes in internal memory, but is quite bad on a DAM. The reason is that when we introduce a unary node into an edge, we have to retrieve the next character along that edge. This will generate a page fault, in the worst case, giving us an I/O bound of $O(N)$, all of which could be random.

An inefficient DAM algorithm: The idea is to introduce in both trees unary nodes for all possible L -lengths which occur in either tree s_o and s_e . If we retrieve the first characters on all the new edges, then we have a simplified problem whose size is now $O((|s_e| + |s_o|)^2)$.

Indeed, we are interested in merging two (compacted) tries where we have no edges to split due to the introduction of unary nodes, and we only need to look at the first character on each edge in order to determine if the edges should be merged. Therefore, we have reduced the problem of merging compacted tries into one of merging uncompactd tries, which we can do by Theorem 3.4. So the question is, how many unary nodes do we need to introduce, and therefore how many characters do we need to retrieve?

All we know is that we will end up merging a single path p_o in s_o with a single path p_e in s_e . We certainly need to introduce a node into p_o for each node in p_e , and vice versa. However, we don't know p_o or p_e , so we need to break every edge in s_o at every possible relevant length. Similarly, we need to break every edge in s_e . In particular, we will break every edge at all L -lengths of any node in both s_e or s_o . So an edge of s_o will also be split according to L -lengths of nodes in s_o itself, and symmetrically for s_e . There are $O(|s_e|)$ L -lengths of nodes in s_e and $O(|s_o|)$ L -lengths of nodes in s_o , and so every edge in either s_e or s_o can be split by $O(|s_e| + |s_o|)$ unary nodes in the worst case. Hence, the overall size of the two side trees will be $O((|s_e| + |s_o|)^2)$ after the introduction of unary nodes.

We can therefore implement the inefficient merging algorithm described above in the I/O complexity of sorting $O((|s_e| + |s_o|)^2)$ items per side tree pair, and in the I/O complexity of sorting $\Theta(N^2)$ items overall.

An efficient DAM algorithm: We will apply the quadratic I/O algorithm on small subtrees to find approximate pull nodes. Then we will simulate the internal memory algorithm to reduce the problem size, after which we will repeat the procedure on the smaller trees. Each iteration will take sorting I/O complexity, and we will show that the loop terminates in $O(1)$ rounds, after which we will have our pull node pair descending from the paths p_e and p_o containing the merge nodes.

We initially set $\hat{s}_e = s_e$ and $\hat{s}_o = s_o$. During each iteration of the loop (described below), the trees \hat{s}_e and \hat{s}_o will shrink but will maintain the invariant that they will always contain the pull nodes we are searching for.

Each iteration of the loop consists of the following. Take every $\sqrt{|\hat{s}_o|}^{th}$ leaf in \hat{s}_o and every $\sqrt{|\hat{s}_e|}^{th}$ leaf in \hat{s}_e , and form the skeleton tries \hat{s}'_o and \hat{s}'_e from these leaves (Theorem 3.3). Run the quadratic DAM algorithm on these trees, and produce a pair of "pull" nodes l_o and l_e . However, these need not be true pull nodes of \hat{s}_o and \hat{s}_e , respectively, since they were found by merging sampled trees, rather than the trees themselves. However, they are useful for finding true pull nodes as follows. Mimic the internal memory algorithm by performing a *retracing step*, in which the paths \hat{p}_e and \hat{p}_o leading to l_e and l_o are traced in \hat{s}_e and \hat{s}_o . Merge these two paths by retrieving the needed splitting characters in a batch of size $O(|\hat{s}_e| + |\hat{s}_o|)$ until we reach the end condition, that is, we find a node, say w.l.o.g. $u_e \in \hat{p}_e$, that splits an edge in \hat{p}_o and whose outgoing edge in \hat{p}_e has first labeling character different from the corresponding one lying on \hat{p}_o . We then insert a unary node u_o at this edge on \hat{p}_o , with $L(u_o) = L(u_e)$, and treat u_o and u_e as "anchors." These anchors may have only one descending side tree pair, and hence we detect it by checking if there is a pair of edges outgoing from the anchors which start with the same character. If so, we set \hat{s}_o and \hat{s}_e equal to that pair, and we recurse on this pair. Otherwise, l_e and l_o is a pull pair.

The correctness of the algorithm follows from the correctness of the internal memory algorithm, since the retracing step does exactly the same side tree manipulations that we are doing in the internal memory algorithm. The running time follows from the fact, which we will show below, that after the first iteration

the new size of either \hat{s}_o or \hat{s}_e is the square-root of its original value.

Lemma 4.6 *After each iteration of the loop, the new size of either \hat{s}_o or \hat{s}_e is the square-root of its original value.*

Proof: Given the “sampled” pull nodes $l_e \in \hat{s}'_e$ and $l_o \in \hat{s}'_o$, the algorithm traces in the full side trees \hat{s}_e and \hat{s}_o , the paths \hat{p}_e and \hat{p}_o leading to l_e and l_o . It then merges these two paths until the end condition is reached, that is, a node, say w.l.o.g. $u_e \in \hat{p}_e$, is found that splits an edge in \hat{p}_o but the next character, say c_o , on this edge does not match the one labeling the edge in \hat{p}_e outgoing from u_e , say c_e , if any. There are three cases and in all of them either we find the pull nodes, or we significantly shrink one of the side trees, either \hat{s}_o or \hat{s}_e , by the required amount:

- Case u_e is a leaf. Here c_e does not exist. Then u_e is actually a pull node. The other pull node in s_o can be taken as any of the leaves descending from the edge split by u_e .
- Case u_e occurs also in the skeleton tree \hat{s}'_e . Since we met the end condition at u_e when retracing \hat{p}_e , we can immediately conclude that we also stopped at u_e when merging the two skeleton trees \hat{s}'_e and \hat{s}'_o . Consequently no edge in \hat{s}'_e outgoing from u_e is labeled by c_o . Therefore, let us denote by e' and e'' the two adjacent edges in \hat{s}'_e outgoing from u_e such that their first labeling characters c', c'' satisfy the relation $c' < c_o < c''$ (possibly one of them is missing). The algorithm selects as new side tree, the subtree descending from the edge of u_e labeled with c_o . If this edge does not exist, the algorithm correctly selects as a pull node any leaf descending from u_e (and easily derives the other pull node). Otherwise, if this edge does exist then the size of the selected subtree will be $O(\sqrt{|\hat{s}'_e|})$. Indeed, its set of leaves will be contained in the set of leaves of \hat{s}_e which lie between the rightmost leaf descending from e' and the leftmost leaf descending from e'' . There are $O(\sqrt{|\hat{s}'_e|})$ such leaves.
- Case u_e does not occur in the skeleton tree \hat{s}'_e . Hence u_e lies on an edge, say ed , of the skeleton tree \hat{s}'_e (which actually represent a sub-path of \hat{s}_e). Let us denote by L the set of $\sqrt{|\hat{s}'_e|}$ leaves in \hat{s}_e which lie to the right (resp. left) of the rightmost (resp. leftmost) leaf descending from the edge ed if $c_e < c_o$ (resp. if $c_e > c_o$). The algorithm will select one new side tree as the subtree

in \hat{s}_e that descends from the edge of u_e labeled with c_o . This subtree will be either empty, and thus the selected pull nodes are correct, or will be formed by leaves contained in L , and thus it will have size $O(\sqrt{|\hat{s}'_e|})$. ■

Then, in the recursive step, we sample the leaves of the new \hat{s}_o and \hat{s}_e more densely, and thus get at most 3 recursive levels to complete the computation. We can therefore conclude that:

Lemma 4.7 *All the pull nodes in T_e and T_o can be determined in sorting I/O complexity.*

Proof: At each iteration of the loop, we retrieve a number of characters linear in the size of the side tree pair, that is a batch of size $O(|\hat{s}_e| + |\hat{s}_o|) = O(|s_e| + |s_o|)$. Retracing and merging the current side trees \hat{s}_e and \hat{s}_o has the I/O complexity of sorting $O(|\hat{s}_e| + |\hat{s}_o|) = O(|s_e| + |s_o|)$ items. The loop is executed at most 3 times so that the I/O-cost for finding the pull nodes in s_e and s_o is the sorting I/O for $O(|s_e| + |s_o|)$ items. The global algorithm operates simultaneously on all the side tree pairs, thus its I/O-cost is the one stated in the lemma because the total size of all side trees is actually $O(N)$. ■

We are therefore ready to conclude with our main result:

Theorem 4.8 *Given an arbitrary string $S[1, N]$, the suffix tree T_S of S can be constructed in sorting I/O complexity and linear space.*

Proof: Let $C(N)$ be the number of I/Os taken by our algorithm to find T_S . We review the algorithm: (1) Find the odd tree T_o and the even tree T_e , by Lemmas 3.6 and 3.7; (2) Find the anchor pairs, by Lemma 4.3, and decompose T_o and T_e into side tree pairs, by Lemma 4.4; (3) For each side tree pair, find a pair of pull nodes, by Lemma 4.7; (4) From the pull nodes, compute the merge nodes, by Lemma 4.5; (5) Merge T_o and T_e into T_S , using the merge nodes, by Lemma 4.1.

The first step takes $C(n/2)$ plus sorting I/O complexity, and the rest has sorting I/O complexity, thus establishing the theorem. ■

5 Conclusion

We have shown how to reduce, more or less, suffix tree construction to sorting and scanning, thus showing that the random—total page I/O profile of suffix tree construction matches that of sorting. There are a number of issues associated with memory bottlenecks, both in general and for string processing, which remain to be investigated.

For example, sorting has an uninteresting tradeoff between random and total page accesses. Are there other natural problems with more interesting tradeoffs, for example, where a few random page accesses yields a substantial savings in total page accesses? Are there parameter-free formalizations of random versus block I/O effect which provide greater descriptive power for comparing algorithms?

In this paper, we have worked with a standard assumption in this area, namely, that we can specify which pages will be evicted from the main memory at each step. It would be worthwhile to consider a model in which some fixed caching scheme manages page evictions. Many issues need to be formalized, since in the current cost model of counting page I/Os, one could influence the caching algorithm unfairly to evict desired pages at no additional cost.

Acknowledgement

We'd like to thank Bruce Hillyer for many helpful discussions.

References

- [1] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 204–216, 1987.
- [2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, pages 1116–1127, 1988.
- [3] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, 1990.
- [4] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory. In *Proc. of the 29th Ann. ACM Symp. on Theory of Computing*, pages 540–548, 1997.
- [5] Y. Chiang, M.T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. *Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.
- [6] M. Dahlin. Trends in disk technology. <http://www.cs.berkeley.edu/~dahlin/techTrends>, 1998.
- [7] M. Farach. Optimal suffix tree construction with large alphabets. *Proc. of the 38th IEEE Annual Symp. on Foundation of Computer Science*, pages 137–143, 1997.
- [8] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. *Proc. of 23rd International Colloquium on Automata Languages and Programming*, pages 550–561, 1996.
- [9] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search (extended abstract). In *Proc. of the 27th Ann. ACM Symp. on Theory of Computing*, pages 693–702, 1995.
- [10] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Addison Wesley, 1998.
- [11] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- [12] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.
- [13] Y. N. Patt. The I/O subsystem—a candidate for improvement. *IEEE Computer*, 27(3), 1994.
- [14] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.
- [15] E. Shriver. *Performance modelling for realistic storage devices*. PhD thesis, Department of Computer Science, New York University, New York, NY, 1997.
- [16] J. Vitter. External memory algorithms. Invited Tutorial, 17th Ann. ACM Symp. on Principles of Database Systems (PODS '98), Seattle, WA, 1998, 1994.
- [17] J. Vitter and E. Shriver. Algorithms for parallel memory: Two-level memories. *Algorithmica*, 12:110–147, 1994.
- [18] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.